L10: Entry 12 of 18 File: USPT Oct 7, 1997

DOCUMENT-IDENTIFIER: US 5675796 A
TITLE: Concurrency management component for use by a computer program during the transfer of a message

Application Filing Date (1):
19960816

Brief Summary Text (6):
FIG. 2 depicts a flowchart of the steps performed for a typical RPC. The calling process 106 first specifies which remote procedure 110, 112, 114 of the remote process 104 to invoke (step 202). After specifying which procedure to invoke, the calling process 106 provides the appropriate parameters for that procedure in a call. Next, the calling process 106 invokes the RPC facility 108 and passes the RPC facility a procedure identifier for the procedure to be called together with the parameters for that procedure (step 204). The RPC facility 108 receives the procedure identifier and the forwarded parameters, locates the requested procedure within the remote process 104, and invokes the requested procedure with the forwarded parameters (step 206). The remote procedure processes the received parameters and returns the results of the procedure call to the RPC facility 108 (step 208). The RPC facility 108, in turn, returns the received results from the remote procedure to the calling process 106 (step 210). While the RPC facility 108 invokes the remote procedure 110, 112, 114, the calling process 106 is unable to perform any other operations, thereby being "blocked."

Brief Summary Text (7):
There are two methods by which an RPC can be invoked: asynchronously and synchronously. A synchronous invocation of a remote procedure refers to when the calling process 106 is blocked while the RPC is processing (as described in FIG. 2). The alternative approach to a synchronous RPC is an asynchronous RPC. An asynchronous RPC is one in which the execution of the calling process and the RPC are not synchronous. An asynchronous RPC allows the calling process 106 to perform other operations while the RPC is processing ("pending"). When a calling process 106 invokes an RPC asynchronously, the calling process 106 may perform other operations; thus providing for more efficient operation of the calling process 106. When the calling process 106 performs other operations, the calling process 106 may perform operations on a separate path of execution ("thread"). When a calling process 106 can perform multiple paths of execution, the calling process 106 is known as being multi-threaded. Although performing an asynchronous RPC increases the efficiency of a calling process 106, there are problems associated with asynchronous RPCs.

Brief Summary Text (8):
One problem associated with asynchronous RPCs is that the calling process 106 must perform concurrency management. Concurrency management refers to managing multiple threads so that each thread may concurrently execute in a reliable and robust manner. For example, when a calling process 106 has many asynchronous RPCs pending at once, the calling process 106 must ensure that the returned results from one RPC are returned to the proper thread and that the threads do not cross. In addition, when a calling process 106 can accept RPCs from other processes, each incoming RPC

requires a separate thread to be forked and requires concurrency management of the threads. If the threads for either outgoing or incoming RPCs cross, the calling process 106 operates in an undesirable manner. Therefore, each process that performs asynchronous RPCs is responsible for performing concurrency management.

Brief Summary Text (9):
Another problem with performing asynchronous RPCs is that the calling process 106, bears the burden of determining the location of the procedure (i.e., whether the procedure is local or remote) to be called. Here, the mechanics of the RPC are not transparent to the calling process 106. For example, suppose that a calling process 106 sends an RPC request and then performs other operations while waiting for a reply to the pending RPC. However, if the RPC is instead local to the calling process 106, the procedure call executes and returns instantaneously, and the calling process 106 is not capable of handling the results from the procedure call at that time. Therefore, since a calling process 106 must know the location of the procedure, the calling process 106 has less flexibility when using an RPC facility 108. In addition, the code of the calling process 106 is more complex since the calling process 106 must know the location of the procedures the calling process wishes to invoke.

Brief Summary Text (10):
The second method by which an RPC can be invoked is a synchronous method. The steps performed in synchronous operation of RPCs is described in the text associated with FIG. 2. Although synchronous operations of an RPC avoids the problems inherent with the asynchronous operations of an RPC, the resources of the calling process 106 are used less efficiently. In addition, when a calling process 106 invokes RPCs synchronously, the calling process 106 is prone to deadlock. Deadlock refers to a situation in which two processes are at a standstill waiting for each other to perform some operation. Since each process is waiting on the other, neither process is able to perform an operation and, thus, both processes are put in a state of perpetually waiting for the other process. For example, the problem of deadlock may arise when a calling process 106 is performing a synchronous RPC and is blocked while the RPC is pending. While the RPC for the calling process 106 is pending, the remote process 104 may be waiting on a response to an RPC or other communication that the remote process 104 has sent to the calling process 106. Since the calling process 106 is blocked, both the calling process 106 and the remote process 104 are deadlocked.

Detailed Description Text (2):
The preferred embodiment of the present invention provides an improvement over the prior art for performing RPCs. The preferred embodiment of the present invention provides a concurrency management mechanism for a calling process that wishes to make an RPC. Therefore, since a calling process does not need to perform concurrency management, the code of the calling process is simplified. In addition, the preferred embodiment of the present invention provides a method and system for communication between processes that is as robust as synchronous RPC, yet as flexible as asynchronous RPC. The method and system for communication of the preferred embodiment is provided through a robust interface and a registration mechanism that are flexible enough to perform concurrency management for any process. Therefore, the preferred embodiment of the present invention leads to a more efficient use of the resources of a calling process and to an increase in system throughput and reliability.

Previous Doc Next Doc Go to Doc#

L7: Entry 5 of 63 File: USPT Apr 6, 2004

DOCUMENT-IDENTIFIER: US 6718399 B1
**** See image for Certificate of Correction ****
TITLE: Communications on a network

Application Filing Date (1):
19960610

Detailed Description Text (19):
Server available data path 26 being currently described operates when the server object is available. When a server object is available within a conventional external RPC which operates in a manner similar to that of the inter-platform portion of the OPM facility of the present invention as shown in FIG. 2, the server object listens for message calls to it using a conventional asynchronous RPC signal.

Detailed Description Text (20):
In a similar manner as indicated in step 36, server object 16 listens to message queue 22, whenever server object 16 is not unavailable, using a conventional asynchronous RPC signal to determine by means of object handles related thereto in message queue 22 when a message call has been placed in message queue 22 for server object 16. As further indicated by step 36, when server object 16 determines that a message call to server object 16 has been placed in message queue 22, server object 16 causes that message call to be transferred to server object 16. As indicated by step 38, the communications between server object 16 and client object 14 occur through message queue 22. In particular, the acknowledgement of receipt of the message call is sent by server object 16 to client object 14 through message queue 22. If appropriate, data may then be transferred through the queue between the client and server objects.

Detailed Description Text (26):
Server unavailable data path 40 is simply the same path as server available data path 26 except that it operates when the server object is not available. Client call path 42 is the client object to message queue portion of server unavailable data path 40. In particular, step 34 operates to place a message call from client object 12 onto message queue 22 via client call path 42. As noted above with respect to step 36, a server object listens for message calls to it using a conventional asynchronous RPC signal. However, if server object 16 is not available when called by client object 12 via client call path 42 because--for example--server object 16 is serving client object 14 via server available data path 26, server object 16 is not then listening for message calls.

Detailed Description Text (28):
In particular, as indicated in step 36, server object 16 listens to message queue 22 when server object 16 becomes available. Server object 16 uses a conventional asynchronous RPC signal via server acknowledgement and listening path 44 to determine by means of object handles related thereto in message queue 22 when a message call has been placed in message queue 22 for server object 16. As further indicated by step 36, when server object 16 determines that a message call to server object 16 has been placed in message queue 22, server object 16 causes that

message call to be transferred to server object 16. As indicated by step 38, the communications between server object 16 and client object 14 occur through message queue 22. In particular, the acknowledgement of receipt of the message call is sent by server object 16 to client object 14 through message queue 22. If appropriate, data may then be transferred through the queue between the client and server objects.

Other Reference Publication (4):
L. E. Heindel, et al, "Highly Reliable Synchronous and Asynchronous Remote Procedure Calls", IEEE, pp. 103-107, May 1996.

Previous Doc Next Doc Go to Doc#

L10: Entry 6 of 18 File: USPT Jun 26, 2001

DOCUMENT-IDENTIFIER: US 6253252 B1
TITLE: Method and apparatus for asynchronously calling and implementing objects

Application Filing Date (1):
19960711

Detailed Description Text (20):
The code generator 111 generates a stub function to handle asynchronous object implementation. The asynchronous server stub function is similar to the synchronous stub function, in that the asynchronous stub function receives (from the server application) the implementation handle and the address of a method to up-call from the server application. The asynchronous stub function, in turn, calls CEE_SET_METHOD, which connects the server application method to the operation corresponding to the stub. This call to CEE_SET_METHOD also notifies the CEE (via a parameter to CEE_SET_METHOD) that the method is to be implemented asynchronously. Since the up-called server application method is asynchronous, the CEE will respond to the client upon a call to a generated response stub function also contained in the server stub file 89. The generated response stub function receives the operation's output parameters and a call identifier. These parameters are passed to a CEE function, CEE_RESPOND (described below), which responds to the client application.

Detailed Description Text (31):
To implement the object synchronously, the server-side CEE 85 calls the appropriate method in the server application (as specified by the initialization routine's call to a server stub function which, in turn, called CEE_SET_METHOD). Preferably, the server-side CEE 85 provides three parameters to the method in the server application: (1) An optional call identifier that is used by the server application method to allocate and deallocate memory for the call; (2) An exception identifier to hold error information that can be returned to the caller; and (3) The input parameters to the requested operation in the order as they were originally defined in IDL. The method uses the input parameters and carries out the request in step 509. The call identifier can be used to call other CEE functions which automatically allocate and deallocate memory within the server application method, as described below in connection with FIG. 10.

Detailed Description Text (35):
Next, in step 605, the object is called using a generated asynchronous client stub function. This generated asynchronous stub function receives the object's proxy handle, the input parameters to the operation, a call tag (described below) and the address to a "completion routine" within the client application. The completion routine will be called by the client-side CEE when a response to the object call has returned. The asynchronous stub function, in turn, down-calls a client-side CEE function, CEE_OBJECT_CALL_POST, which calls the object. CEE_OBJECT_CALL_POST is defined in C as follows:

Detailed Description Text (36):
The stub function provides the proxy handle for the requested object and an operation index that specifies which of the object's operations is to be performed.

The param_vector parameter supplied here is an array containing the addresses of the object call's input parameters only. The input parameters are stored in the array in the same order as they were defined in IDL to permit object calls across multiple platforms. The call_tag1 parameter is a constant used to identify this call to the object.

Detailed Description Text (37):
The asynchronous client stub function and CEE_OBJECT_CALL_POST also receive the address of a "completion routine" in the client application that will be called by the client-side CEE 75 when a response to the object is returned to the client application or an error condition (or other exception) has been received by the client-side CEE. In step 607, the client-side CEE stores the completion routine address in the proxy handle for later use. When the call returns for a particular proxy handle, the client-side CEE will extract the completion routine address from the proxy structure and call the completion routine. The completion routine will be discussed below. If multiple calls are made requiring the same completion routine, the call_tag1 parameter may be used to identify a particular call within the completion routine. The call_tag1 parameter is also stored in the proxy structure in step 607.

Detailed Description Text (40):
The client-side CEE locates the proxy handle structure for the transmitted response in the client CEE capsule. In step 617, the client-side CEE extracts the completion routine address and call_tag1 identifier from the proxy structure. The client-side CEE calls the completion routine in the client application, in step 619. The client-side CEE provides the completion routine with the exception information, the output parameters of the object's methods and an optional identifier tag (as specified above by call_tag1) to identify which asynchronous call has completed (if multiple calls use the same completion routine). The completion routine in the client application can then use these parameters as necessary.

Detailed Description Text (52):
The asynchronous method performs its designated function. When completed, the asynchronous method calls a response function in step 811. The asynchronous method passes the context parameter containing the call identifier and output parameters (as well as any other context that may be useful) to the response function. The response function, in turn, calls the asynchronous response stub function in the server application in step 813. As discussed above, the response stub function contains a down-call to CEE_RESPOND. If the original method had not called an asynchronous method, but was specified as an asynchronous method in the initialization routine, the original method could have called the asynchronous response stub function directly. Alternatively, any method in the server application can call CEE_RESPOND. CEE_RESPOND is defined in C as follows:

Detailed Description Text (54):
In step 815, the response is sent back to the client-side CEE via interface 84. The client-side CEE up-calls the appropriate method in the client application and provides the output parameters and exception information from the call to CEE_RESPOND.

Detailed Description Text (56):
The server-side CEE 85 calls the server application's initialization routine when the implementation libraries are loaded. The routine, beginning at line 901, obtains an interface handle by calling a server stub function at line 902. The interface handle is used to create an implementation handle at line 903 which, in turn, is passed to a server stub for setting the address for a method in the server application. At line 904, the address for the server method, NowMethod, is specified as the method to be up-called by the server-side CEE for the Tim_Now operation. Moreover, by calling an asynchronous stub function to set the address, server application notifies the server-side CEE that a call to CEE_RESPOND is

required before responding to the client (rather than responding automatically upon exiting the up-called method). Further, the server-side CEE 85 will pass a call identifier into the method to identify the call.

Detailed Description Text (63):
The call_id parameter identifies the particular call so that the CEE can automatically deallocate the memory upon completion of the call. The len parameter specifies the number of bytes to allocate. The CEE returns the address of the allocated memory using the ptr parameter.

Other Reference Publication (2):
Lin et al; "an asynchronous remote procedure call system for heterogeneous programming", IEEE Electronic library, 1991.*

Previous Doc Next Doc Go to Doc#

L10: Entry 8 of 18 File: USPT Mar 9, 1999

DOCUMENT-IDENTIFIER: US 5881315 A
TITLE: Queue management for distributed computing environment to deliver events to interested consumers even when events are generated faster than consumers can receive

Application Filing Date (1):
19950818

Brief Summary Text (12):
It is still another object of the present invention to provide a DCE RPC-based asynchronous event management service wherein event suppliers and consumers are authenticated prior to using the service by a DCE security service.

Detailed Description Text (12):
Event Type Database 44 stores information used by EMS and event suppliers and consumers to generate "event types," each of which are a class of events that have the same event type format. An event type format is described via an "event type schema" and identified by a unique universal identifier (UUID). An event type schema is a description of an event type and consists of a list of attribute name/type pairs which specify the data format of an event. An attribute name is a string that uniquely identifies an attribute of a given event type. An attribute type is the data type of an event attribute, which defines the format of the data in the attribute. Event type schema data (stored in database 44) and/or event header information is used by an event consumer to construct an "event filter group" to tell EMS which events to forward to that consumer.

L10: Entry 17 of 18 File: USPT Jul 30, 1991

DOCUMENT-IDENTIFIER: US 5036459 A
TITLE: Multi-processor computer system with distributed memory and an interprocessor communication mechanism, and method for operating such mechanism

Application Filing Date (1):
19890309

Detailed Description Text (20):
To abstract from low level machine details we introduce a language construct called Remote Asynchronous Call (RAC). In this section we give a sketch of its implementation. The syntax of RAC is as follows:

Detailed Description Text (21):
The execution of a RAC first evaluates the parameter expressions of the procedure call. Then the node expression is evaluated, next an asynchronous message containing the parameters and procedure name is constructed and sent to that node. The destination node executes the involved procedure. The process that executes the RAC continues its operation immediately after sending the message.